SOCKMAP: The ultimate weapon

In recent years Linux Kernel introduced an eBPF virtual machine. With it, user-space programs can run specialized, non-turing-complete bytecode in the kernel context. Nowadays it's possible to select eBPF programs for dozens of use cases, ranging from packet filtering, to policy enforcement.

From Kernel 4.14 Linux got new eBPF machinery that can be used for socket splicing - SOCKMAP. It was created by John Fastabend at Cilium.io, exposing the Strparser interface to eBPF programs. Cilium uses SOCKMAP for Layer 7 policy enforcement, and all the logic it uses is embedded in an eBPF program. The API is not well documented, requires root and, from our experience, is slightly buggy. But it's very promising. Read more:

- LPC2018 - Combining kTLS and BPF for Introspection and Policy Enforcement Paper Video Slides
- Original SOCKMAP commit

This is how to use SOCKMAP: SOCKMAP or specifically "BPF_MAP_TYPE_SOCKMAP", is a type of an eBPF map. This map is an "array" - indices are integers. All this is pretty standard. The magic is in the map values - they must be TCP socket descriptors.

This map is very special - it has two eBPF programs attached to it. You read it right: the eBPF programs live *attached to a map*, not attached to a socket, cgroup or network interface as usual. This is how you would set up SOCKMAP in user program:

```
sock_map = bpf_create_map(BPF_MAP_TYPE_SOCKMAP, sizeof(int), sizeof(int), 2, 0)

prog_parser = bpf_load_program(BPF_PROG_TYPE_SK_SKB, ...)
prog_verdict = bpf_load_program(BPF_PROG_TYPE_SK_SKB, ...)
bpf_prog_attach(prog_parser, sock_map, BPF_SK_SKB_STREAM_PARSER)
bpf_prog_attach(prog_verdict, sock_map, BPF_SK_SKB_STREAM_VERDICT)
```

Ta-da! At this point we have an established `sock_map` eBPF map, with two eBPF programs attached: parser and verdict. The next step is to add a TCP socket descriptor to this map. Nothing simpler:

```
int idx = 0;
int val = sd;
bpf_map_update_elem(sock_map, &idx, &val, BPF_ANY);
```

At this point *the magic happens*. From now on, each time our socket `sd` receives a packet, prog_parser and prog_verdict are called. Their semantics are described in the strparser.txt and the introductory SOCKMAP commit. For simplicity, our trivial echo server only needs the minimal stubs. This is the eBPF code:

```
SEC("prog_parser")
int _prog_parser(struct __sk_buff *skb)
{
    return skb->len;
}

SEC("prog_verdict")
int _prog_verdict(struct __sk_buff *skb)
{
    uint32_t idx = 0;
    return bpf_sk_redirect_map(skb, &sock_map, idx, 0);
}
```

Side note: for the purposes of this test program, I wrote a minimal eBPF loader. It has no dependencies (neither bcc, libelf, or libbpf) and can do basic relocations (like resolving the `sock_map` symbol mentioned above). See the code.

The call to `bpf_sk_redirect_map` is doing all the work. It tells the kernel: for the received packet, please oh please *redirect* it from a receive queue of some socket, to a transmit queue of the socket living in sock_map under index 0. In our case, these are the same sockets! Here we achieved exactly what the echo server is supposed to do, but purely in eBPF.

This technology has multiple benefits. First, the data is never copied to userspace. Secondly, we never need to wake up the userspace program. All the action is done in the kernel. Quite cool, isn't it?

We need one more piece of code, to hang the userspace program until the socket is closed. This is best done with good old `poll(2)`:

```
/* Wait for the socket to close. Let SOCKMAP do the magic. */
struct pollfd fds[1] = {
    {.fd = sd, .events = POLLRDHUP},
};
poll(fds, 1, -1);
```

Full code.